Scalable Feedback for Student Live Coding in Large Courses Using Automatic Error Grouping

Sven Strickroth
sven.strickroth@ifi.lmu.de
LMU Munich
Munich, Germany

ABSTRACT

Programming courses in higher education are often attended by several hundred students. In such large-scale courses, direct instruction is often the last resort, resulting in mostly passive students and limited social interaction. The instructor may present worked examples or perform live coding and the students try to reproduce these on their devices. However, there is usually no live coding where students work on small programming assignments themselves directly in class, because the instructor does not have a timely/rapid overview of the most important common issues that are prevalent in class to support the students. This paper presents experiences with a teaching format to activate students that addresses the aforementioned issues. After students have worked on a small assignment and uploaded their solution attempt within a specified time period, an extended e-assessment system analyzes all submissions and instantly provides an overview of the number of correct submissions as well as all common errors and their frequency to assist the instructor in the immediate tailored discussion. This approach makes it possible to engage students, make student performance visible to all participants, and discuss the most common errors. Students liked "their" live coding, the discussion of "their" errors, and want to do it more often, although not many students uploaded their solution attempts. The teaching scenario, benefits, pitfalls, possible improvements, and further application scenarios are discussed.

CCS CONCEPTS

• **Applied computing** → **Education**; • **Social and professional topics** → **Computing education**.

KEYWORDS

Live Coding, Audience Response, Just-in-Time Teaching, Programming Education, Live Feedback, Formative Assessment, E-Assessment

ACM Reference Format:
Sven Strickroth. 2024. Scalable Feedback for Student Live Coding in Large Courses Using Automatic Error Grouping. In *Proceedings of the 2024 Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2024), July 8–10, 2024, Milan, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3649217.3653620

ITiCSE 2024, July 8–10, 2024, Milan, Italy

1 MOTIVATION

Many beginners perceive learning programming as hard and struggle with it [23]. Programming requires several skills to be mastered at the same time such as knowledge of the formal programming language, problem-solving skills, computational thinking, and debugging skills [10, 23].

In higher education, programming (concepts) are typically taught in lectures with accompanying homework assignments to practice the introduced concepts and exercise sessions led by teaching assistants. In these courses, instructors often have to deal with high heterogeneity as some students have already learned programming in school or on their own and such courses are regularly attended by a variety of academic disciplines such as engineering, science, or economics [33]. At the same time, introductory programming courses are often attended by several hundred students [5, 38]. In such scenarios, direct instruction is often the last resort, resulting in limited social interaction, discussion, and personal feedback, which ultimately leads to passive students and higher dropout rates [5, 38]. Besides direct instruction of concepts, live coding led by the teacher is often used as a teaching method to demonstrate solution strategies [34]. Students can be encouraged to reproduce the steps on their devices but this is not the same as solving the problem on their own. Furthermore, experience shows that often only the "better" students ask questions and the pace of the lecture is adjusted to suit them. In general, many students are afraid to ask (potentially "stupid") questions in front of a large audience, or are intimidated by detailed questions from more advanced students [15, 42].

Actively working on problems yourself and receiving feedback is considered one of the most important drivers of learning [13]. However, providing rapid/timely (personal) feedback is difficult to do in large courses. This is particularly true for in-class coding assignments: Without having an overview of issues that affect many students, it is not possible to discuss the actual most pressing issues. Instructors may discuss alleged frequent mistakes, however, these often do not match the actual encountered ones [7, 18]. Homework assignment may also be not optimal, because they have to be handed in before a deadline, and then corrected. By the time the feedback is available to the students and to the instructor, it is often at least a week since the concept was taught in a lecture.

Educational software can help solve these problems: enable new teaching scenarios, activate students, provide instructors with more information about their students, and promote interactivity and discussion [38]. The contribution of this paper is twofold: First, it describes in detail a teaching format in which students are activated by working directly in class on small programming assignments and instructors are supported with automatically generated instant overviews of student performance and the most common errors

and their frequency for just-in-time teaching (i. e., to scale student live coding and enable instructors to provide feedback on the most common syntactic and functional errors).[1] Second, the paper presents experience gained with this approach in a first-semester Java computer science course.

The remainder is structured as follows: First, related research is discussed. Second, the goals and teaching scenario are described. Third, the gained experiences are presented. The paper concludes with a discussion, summary, and an outlook.

2 RELATED RESEARCH

Four fields of related research need to be considered: First, live coding in lectures, audience response systems (ARS), automatic e-assessment systems for evaluating programming assignment submissions, and research on programming errors.

There is a recent literature review on live coding by Selvaraj et al. [34]. The authors provide a common base definition of live coding, stating that "it is the process of designing and implementing a coding project in front of a class during lecture" [34, p. 166], but conclude that there is disagreement about the minimum requirements: They found that live coding most often refers to an instructor-led activity where code is written from scratch and/or prepared code is discussed. Student-led and collaborative live coding, where instructors take up students' ideas and/or students program alongside instructors, is less common (e. g., [19, 32]). It remains unclear, however, whether these approaches scale. The authors conclude that live coding is rarely active learning in contrast to short in-class exercises after an instructor's demonstration, where students are not just passively listening [34]. Commonly reported benefits include improved debugging skills, exposure to programming as a process, increased student engagement, and application of concepts. Reported disadvantages of live coding include its time-consuming nature and students struggling to keep up with the pace of programming or to take notes. There are also tools that support instructor-led live coding with special highlighting and sequencing features (e. g., [8]).

An established method to activate students and to get an insight on the understanding of students in (large) classes are audience response systems (ARS), which (anonymously) collect student responses and instantly provide aggregated results [11]. There is a wide variety of didactic scenarios such as knowledge checks of previously taught concepts, feedback, group interactions, competitions, or peer discussion [20, 24–26]. ARS have been shown to increase student engagement, knowledge retention, participation, and classroom interaction (e. g., [3, 11, 17, 20, 24, 25, 29, 30]). In recent years, the development of ARS has been actively pursued (e. g., [12, 22, 24, 42]). ARS are often implemented as smartphone apps or (mobile) websites, allowing remote students to participate in hybrid courses, but most only support simple multiple-choice questions [24]. In the context of programming education, there are four notable approaches: Mader and Bry proposed an ARS that employs puzzles for small coding assignments in Haskell [24]. There are approaches by Hauswirth and Adamoli [14] as well as Robbins [31] to support teaching Java using multiple-choice questions and small fill-in-the-gap programming tasks with a simple text field. Both systems allow the lecturer to manually inspect/select submissions for discussion in class, but there seems to be no automatic evaluation. It is also unclear, how many students participated in the case studies (seems to be less than 20). Finally, there is a prototype of Ebert and Ring [10] that can collect and present common syntax errors of Python assignments in-class. Their tool also has features for sharing code with fellow students and/or the instructor. It was successfully tested during multiple weeks in a course with about 90 students. The authors report that the error information was helpful, but not all submissions could be discussed during the course. Nevertheless, they argue strongly in favor of in-class coding assignments for students as opposed to the presentation of worked examples or live coding sessions conducted by an instructor. However, their system cannot assess functional correctness.

To facilitate the teaching of large classes, many (semi-)automated e-assessment systems have been developed [21, 41]. These tools can deliver rapid feedback without the need for instructor involvement, or support semi-automatic grading (e. g., [39, 40]). However, such systems are neither designed nor optimized for in-class usage.

Extensive research has been conducted on the analysis of misconceptions, (common) errors in programming environments, their frequencies, and origins (e. g., [1, 4, 9, 18]). In general, students seem to frequently make similar errors and, therefore, a grouping makes sense in principle (cf. [16]). Disagreement exists among instructors, as well as discrepancies between their perceptions of frequent programming errors and the actual errors encountered by students [7, 18]. Hence, it makes sense to analyze students' submissions and discuss the actual most common errors. There also is research different categorizations, but these are out of scope (e. g. [1, 27, 43]; runtime errors are rarely considered, e. g., [27]). Awareness of students' errors and misconceptions and strategies to address those is said to be a key element of pedagogical content knowledge [9].

In summary, students should actively solve programming assignments during lectures. But there is a need for "live" overviews of common syntactic *and* functional errors for teachers to effectively discuss the actual most common issues, especially in large classes. Also, little experience exists with such approaches in large courses.

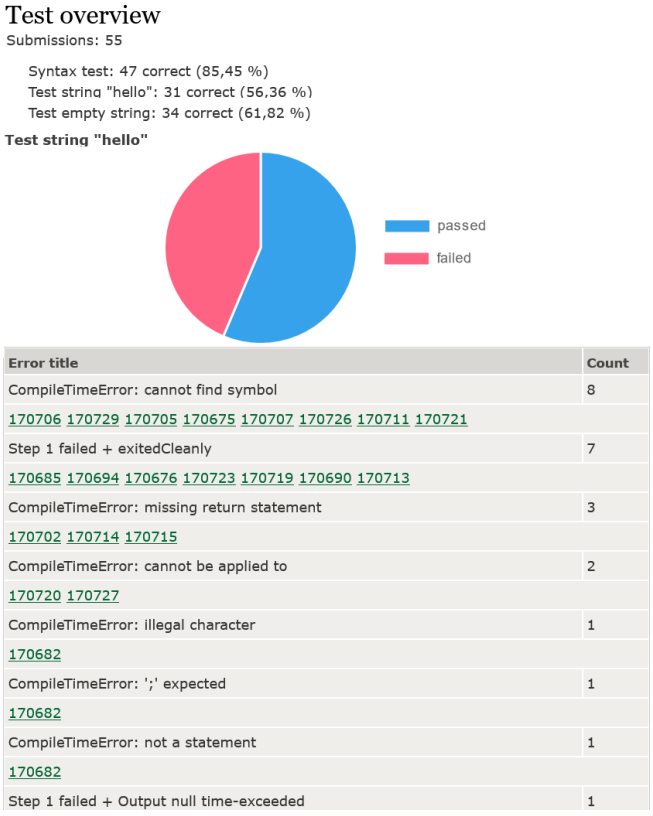
3 GOAL AND TEACHING SCENARIO

The main goal is to activate students and let them work on programming assignments themselves during a lecture. For this to be effective, it is important for the instructor to know what the students are doing and what the common problems are so that s/he can address them directly based on the specific needs of the group. Furthermore, the teaching approach aims to show students alternative solution strategies. This goes beyond the traditional methods of live coding or the presentation of worked examples by an instructor where only a few predetermined strategies are discussed by the lecturer and the students are often rather passive consumers (cf. [10, 34]). In the latter case, students cannot apply their newly acquired knowledge on their own and develop their own possibly creative solutions in an iterative and exploratory process. Involving students in programming tasks during lessons engages them and improves their programming skills. In addition, students can see that they are not the only ones with a particular error and can compare their own performance with that of their peers, giving them a better understanding of their learning progress.

[1] The general idea and the requirements have already been published as a poster: [37]

To implement this teaching approach, students are required to bring their laptops to in-person lectures or use their own (desktop) computers for remote attendance. The lecturer prepares coding problems (code templates, and tests) in advance that need to be completed with simple algorithms related to the concept taught in the lecture. This approach fosters higher-order thinking skills, such as problem-solving, compared to using simplistic multiple-choice questions. These assignments are then presented in class, and students work on them for about 10–15 minutes. Optionally, a teacher-led or collaborative live coding session can be held beforehand in which the specific concept is revisited and/or similar problems are solved that students can built upon. This allows students to observe the process of problem solving and see how to apply the concepts taught before attempting it themselves.

During this assignment period, students have the opportunity to discuss and collaborate with their seat neighbors while the instructor can move around the lecture hall to address questions, listen to arguments, and identify common challenges (cf. peer discussion [26]). Once students complete their work, they upload it to an extended e-assessment system. When the time limit expires, the instructor closes the submission and triggers the automatic evaluation of all submissions.



Error title	Count
CompileTimeError: cannot find symbol	8
170706 170729 170705 170675 170707 170726 170711 170721	
Step 1 failed + exitedCleanly	7
170685 170694 170676 170723 170719 170690 170713	
CompileTimeError: missing return statement	3
170702 170714 170715	
CompileTimeError: cannot be applied to	2
170720 170727	
CompileTimeError: illegal character	1
170682	
CompileTimeError: ';' expected	1
170682	
CompileTimeError: not a statement	1
170682	
Step 1 failed + Output null time-exceeded	1

Figure 1: Screenshot of the test result overview with grouped common errors and links to the corresponding submissions

The submissions are automatically checked in the background using established e-assessment features, and the focus shifts to the features provided by ARS: The system groups submissions by errors and provides an overview of the overall test results as well as a ranking of the errors by frequency for the lecturer, which can be projected for the class to see (cf. Figure 1).

Based on the observations while moving around in the lecture hall and this overview, the instructor gets an impression of the understanding and performance of the class and can start a tailored discussion. The instructor can clarify any errors by selecting and opening specific submissions. Additionally, the instructor can explain potential strategies for resolving the errors. This can be done for all common errors so that students receive timely feedback on their just submitted work. At the same time, the instructor can highlight different solution strategies and address issues s/he observed while moving around. Finally, the instructor can program/show one or multiple model solutions that have not been previously seen.

4 EXTENDING THE GATE SYSTEM

The web-based e-assessment system GATE [40] developed by the author already supports the automatic test execution and optional feedback generation for students. GATE is designed as a submission system. Students can upload their work as many times as they like until the deadline, even if it contains invalid code. Feedback is not provided automatically after each upload, but students must explicitly request it. This design allows for easy configuration of the availability of tests to students and also prevents abuse (cf. [2]). In this scenario, tests are disabled for students in order to gain insight into their “real” competencies. Three types of tests are supported by GATE: syntax tests based on the OpenJDK compiler, JUnit tests, and input/output (IO) tests. For the IO tests the student code is called in several configurable steps, the result is printed to the console by the test driver, and then compared to predefined expected outputs for each step until all steps are executed, an unhandled error occurred, or a timeout is exceeded and the execution is aborted. The requirements for a system for this teaching approach are [37]:

- providing code templates/skeletons to students
- upload and automatic test capabilities for programming assignments (but no instant feedback on upload for students)
- (missing) tests need to be triggered on-demand and finish within a few minutes (all tests results need to be stored)
- the system should provide a quick, overview of the number of submissions that passed the tests and a ranking of the most common errors (in a grouped way with links to submissions, including both syntactic and functional errors) that can be shown on a projector
- allow the instructor to access all (erroneous and correct) submissions in an anonymous way without disclosing specific students for inspection and discussion in class.

Only minor extensions were necessary in GATE: A new option was integrated to close the submissions and to instantly execute all configured tests. After executing the tests, the results are automatically grouped using a text-based grouping approach. This approach considers syntax errors, failed or passed test cases, and any thrown exceptions to group the errors. A paper with a detailed description of the algorithm is currently under review [35]. Finally, an overview feature (cf. Figure 1) and an anonymization feature have been added that displays only submission IDs instead of student names.

5 EXPERIENCES AND LESSONS LEARNED

The following subsections introduce the setting of the course and present the author's experience as the instructor. Finally, a questionnaire was send to all students enrolled in the course to collect their experiences at the end of the semester after the exam.

5.1 The Setting

The prototype was used in a first-semester introductory Java programming course at LMU Munich, Germany, during the winter semester of 2021/2022. The course was delivered in a hybrid format, allowing students to attend lectures either in person or remotely via Zoom. Lecture recordings were also provided to students. About 900 students were enrolled in that course of which 300–400 students attended the course in person regularly.[2]

The course was designed to consist of two parts. The first part comprised traditional lectures that taught theoretical concepts such as variables, procedures, object orientation, inheritance, and recursion. The second part was designed to be more interactive. In this part, concrete (worked) examples were presented and possible alternative solutions discussed. Additionally, there were two forms of live coding. Live coding of the professor and smaller live coding sessions of the students. The lecturer's coding examples were designed in a way that the student should try to reproduce these synchronously on their devices. Therefore, the students were aware that they need a laptop for the face-to-face lecture. In four of the possible eight practice lectures, the proposed feature of the GATE system was used to activate the students with one programming assignment each. The lecturer covered programming concepts through examples and live coding that were relevant to the new problem the students needed to solve afterwards. As time was already mentioned as a common issue (cf. [14, 34]), only one assignment was set for a lecture. All students who attended a lecture (either in person or remotely) could voluntarily take part in the live coding. The students were already familiar with GATE, as it was also used for the voluntary submission of homework, provided automatic feedback on it, and to orchestrate voluntary homework peer (code) reviews (cf. [36]).

The typical steps of the practical sessions are as follows: First, concepts were revisited using (worked) examples. Prepared examples were explained, interactively modified, and debugged, i. e., the instructor deliberately made mistakes or asked how the example could be extended, how the code could be improved, and what alternative solutions may be possible. In two sessions, interactive object-oriented modeling was conducted under the guidance of the instructor and then partially implemented. Similar to cooking shows, in the last session some simple classes (POJOs) were prepared. Partly means that one method was intentionally left out that could be implemented similarly to another presented method and then the students were asked to complete it on their own.

Table 1 provides an overview of the assignments, including the number of submissions and composed groups for the syntax tests. The first assignment required the implementation of two methods to calculate the sum and mean of all entries in an integer array. The second assignment involved filling two methods (hasPositiveBalance and transferMoneyTo) of a bank account class. In the third assignment, all vocals of a string should be counted recursively. In the fourth assignment, the students were asked to implement missing methods to retrieve all available media of a collaboratively designed library for books/CDs/etc. Code templates were provided for all assignments where methods to be implemented were either empty or missing. For some assignments, especially where multiple methods should be implemented, independent tests were implemented that were grouped and displayed separately (one test overview is shown in Figure 1). About 55–65 students submitted their work in the live coding sessions. Syntactical correct were about 60 to 91 % of the submissions. Functional correctness ranges between 7 and 82 % – edge cases were implemented correctly by very few students (e. g., usage of integer division and no handling of an empty array in assignment 2). For all assignments, there were groups found that consisted of between 2.1 and 3.6 submissions on average, as well as up to three groups consisting of more than 10 submissions for the syntax tests.

Table 1: Overview of submissions (S), topics, week of the semester (W), syntactic correctness (SC), the average number of submissions per group (ASG), and number of groups comprising more than 10 submissions (#G>10S)

W	Assignment title	#S	#SC	ASG	#G>10S
2	Array operations	64	91 %	3.6	3
3	Object orientation	65	86 %	2.9	2
5	Recursion	55	85 %	2.1	0
7	OO design	55	60 %	3.4	3

5.2 Lecturer's Experiences

After each lecture, notes on experiences were taken. This section summarizes and reflects on these. The following categories are inspired by the study of Hauswirth and Adamoli [14].

What problems to post? An important point is to develop assignments that match the concepts taught and are also expected to be doable during class. Also, pedagogical aspects need to be considered when designing the assignment and to develop fine grained tests and test-steps that are usable for forming groups. Here, different functional errors need to be anticipated already (such as integer vs. double division, missing null checks, division by zero; cf. misconception research, e. g. [9]). A significant challenge was to create supplementary smaller assignments in addition to the exercise sheets. This turned out not to be feasible or possible for all practical lectures without much repetition and limited resources. Also, different solution strategies needed to be thought of (e. g., switch case vs. if; for vs. for-each vs. Java streams) in case those were not used by the students for discussing those.

Time to allot for working on the assignments and participation: The scheduled time of about 10 minutes for working on the programming assignment was too long for more experienced students and possibly too demanding for less advanced students in the heterogeneous group. It is also crucial to consider the time needed to download and import code templates into the IDE (Eclipse). This was estimated to take less than a minute, however it took significantly longer for less advanced students. Table 1 shows the number of submissions for each live coding assignment. Compared to the

[2]Rough estimate based on the crowdedness of the lecture hall.

overall enrolled and attending students, the numbers seem to be quite low. The submissions were closed after a 10-second countdown in which no student seem to have indicated to need just a little more time for the submission. Hence, waiting for the majority to hand in their solutions as reported in [14] was not possible.

Quality of the groups, and time to allot for discussion: In general, the formed groups were usable for getting an overview, and selecting one or two submissions for in-depth discussion. Interestingly, the submissions with different errors were also so diverse that different solution strategies could be discussed simultaneously without having to actively search for them. The "cannot find symbol" group turned out to be not optimal as the reasons were too diverse (e. g., wrong package, typo in class/method names, students inventing methods). However, there were not many groups and each group only contained a handful of respective submissions; several ones could be opened manually to decide whether they need to be discussed. All syntax errors in the submissions were reported repeatedly for all function tests – this needs to be optimized. Special care on consistency should be taken when conducting collaborative object-oriented modelling as preparation for an assignment: In the last assignment, a method name was written in a different case than the code template, causing syntax errors for students who did not use the provided template. If multiple methods need to be implemented, students may leave a method empty, resulting in syntactically incorrect submissions (i. e., missing a return statement), even if other methods are implemented correctly. It is a didactic decision whether to leave methods empty or fill them with a bogus return statement. Students may also delete a method referenced in the test code, resulting in a compilation or runtime error. To avoid this, Java reflection should be used. Five to ten minutes were allotted for discussion. This proved to be tight, but sufficient to discuss the most common issues and to present a correct solution.

Technical issues: There were no technical issues experienced – neither with the system nor the WiFi in the lecture hall. The test execution and grouping always took less than half a minute.

Benefits: In general, the live coding of the students and using the system to get an "instant" overview of the common issues was experienced as interesting and helpful. Particularly, higher syntactic correctness rates were expected. Although only a fraction of the students submitted, both the instructor and the students eagerly awaited the results when the submission was closed and the evaluation was triggered. This was experienced as a good starting point for discussing errors as the students seemed to be focused and activated. The instructor also perceived more interactivity, such as more (in-depth) questions about specific errors and possible solutions to fix them, compared to his live coding sessions (there also, intended and unintended errors occurred). In particular, the discussion of most submissions with a "missing return statement" error led to some amusement in the class when they turned out to be just the template. It is hard to quantify how helpful walking around in the lecture hall was. Students rarely dared to ask questions (one to three per session). Students needed to be actively asked whether there are problems. Often "no" was answered. However, observing the solution attempts provided insights into the students' thought processes, which served as a solid foundation for further discussion.

5.3 Students' Questionnaire Evaluation

A digital questionnaire was distributed to all students enrolled in the course, requesting their opinion on the live coding sessions after the end-of-semester exam. 140 students answered the questionnaire. These students uploaded a solution attempt in $\bar{x} = 1.07$ live coding sessions (median=m=1, $[0; 4]$). Only 22 students uploaded their solution in at least 3 sessions. Table 2 presents a summary of the results obtained from the seven closed questions. The Likert scale used ranged from 1 (strongly disagree) to 4 (strongly agree). Most students tried to work on their own solution (overall m=4, even many of those who did not upload: $\bar{x}$=3.6, m=3), want to do this procedure more often (m=4), and found the discussion to add value (m=4). There are tendencies that it was experienced as exciting to discuss mistakes and solutions in plenary (m=3), and the discussion of real assignments motivated the students to attend the lecture or watch it live (m=3). The allotted time was perceived as mostly sufficient (m=3). The ratings of students who uploaded their work in ≥ 3 sessions are statistically significantly higher (all $p < .02$, UTest), except for "I exchanged ideas with my seatmates" ($p = .266$).

Table 2: Overview of the students' ratings on the questionnaire divided by uploads in # sessions, Likert scale from 1 (do not agree at all) to 4 (fully agree); $\bar{x}$: mean, m: median

Statement	≤ 2 sessions			≥ 3 sessions		
	$\bar{x}$	m	n	$\bar{x}$	m	n
I always tried to work out my own solution.	3.6	3	105	4.0	4	22
There was enough time to solve the tasks.	2.8	3	104	3.6	4	22
While solving the assignment, I exchanged ideas with my seatmates.	2.7	3	98	2.4	3	21
I found the discussion of the submissions to add value.	3.3	3	101	3.7	4	22
It was exciting to discuss mistakes and solutions in plenary.	3.2	3	98	3.7	4	21
Discussing real assignments motivated me to attend the lecture or watch it live.	2.7	3	91	3.5	4	19
This procedure should be done more often.	3.4	4	93	3.8	4	19

Furthermore, the questionnaire contained two (optional) open-ended questions "If you haven't regularly participated the live coding, why not?" (31 answers) and "Do you have any comments about the live coding?" (25 answers) that were evaluated using Thematic Analysis [6, 28]. For the first question, the main reasons were that there was too little time for working on the assignments (n=8), students could not attend the lecture due to e. g. conflicts with other courses (n=7), the assignment was not clear enough (n=5), seeing no advantage to work on too easy tasks (n=5), and not having a laptop (n=2). One student answered that s/he was distracted and another student reported that s/he was not able to follow the lecture.

For the second question the most frequent aspect was that the students explicitly liked the approach (n=9). Followed by requesting more time (n=5), acknowledging that the approach already takes a lot of time (n=5), the wishes to discuss more alternative solutions (n=4), and to see the test results for their own solution

(n=4). Mentioned once each were that more assignments should be set, the request for more complex assignments and to conduct this more frequently, the tasks were not always clearly enough, all submissions should be made public, too much focus on errors, "nice variation to traditional lectures", more time for the discussion, and one student found the whole approach to be superfluous.

6 DISCUSSION

The approach has shown that student live coding and discussing the actual most common errors is possible in a large class – not only in smaller courses (cf. [10, 14]). However, not many students participated in the submission of solutions in the lectures (cf. Table 1). Some students were unable to attend, either in person or remotely, and a significant number indicated they needed more time. It is unclear whether this was the main reason or whether they were unable to solve the tasks, did not want to submit partial solutions, or simply chose not to submit. It was anticipated that anonymity would reduce barriers to student participation, as demonstrated by ARS interventions. The submission of unaltered template code may suggest that certain students intended to demonstrate that the assignment was beyond their ability. Additionally, some students have indicated explicitly that the assignments were too easy and chose not to work on them. A solid number of submissions is necessary for an adequate discussion of common errors – this was the case on the study. Gamification approaches (e. g., [25]) may help to attract more students. The low submission rate requires further investigation. Nevertheless, the live coding and evaluation activated the students and got them to focus on the discussion. Also, students seemed to have like the approach and requested to conduct it more often. Identified challenges are comparable to those already reported for smaller classes (cf. [14]). Therefore, this research raises pedagogic questions such as what and how many problems to pose, how to deal with the uneven time needed by students (cf. [14]), but also on how to design tasks and to write good tests that elicit distinctive groups for discussion.

The presented approach focused on making common errors visible and discussing them during a lecture. Hence, the focus and type of grouping are targeted to the instructor who explains the issues on specific examples. Students also requested to see more alternative solution strategies and to get feedback for their own submission. Hence, the prototype should be extended e. g. by suggesting correct submissions with different solution strategies, to show the personal test result to the students (it remains unclear now, why the latter was not considered from early on), or to ask students to resubmit an improved version (cf. peer discussion [26]).

The group names are not optimal, particularly if they should be easily interpretable for others. Instead of writing "Step n", titles of the test cases could be used to improve the group names. Still, discussing the errors needs experience from the lecturer on the different meanings of the errors as well as to quickly understand the submitted code. Explanations or another categorization (cf. [1, 43]) may be helpful, particularly, when the groups are made available to the students. This kind of feedback may help them to better recap the instructor's explanation after the lecture and also to provide further feedback on errors that could not be discussed in-class (e. g., due to time reasons) – a problem reported in [10]. Also, a categorization such as "empty submission" should be considered.

The velocity for executing the tests and grouping the submissions might be increased if the tests are directly triggered when the submissions are uploaded. Still, less than one minute for testing and grouping seems to be okay as some details of the task and gathered experience from walking around in the lecture hall can already be discussed in the meantime.

The presented experience is based on a single course at a specific university. The approach was implemented in a Java course, but it can be applied to other programming languages and scenarios as well. Due to resource constraints, the instructor of the course and author of this paper are the same person. Additionally, there was self-selection of students who participated in live coding and uploaded their solutions, as well as those who completed the questionnaire at the end of the semester. This may introduce bias towards more engaged students. However, there may also be a bias in the opposite direction, where high-performing students did not submit their solutions because they perceived the assignments as "too easy". The data suggests that both scenarios are plausible. Finally, there is an overrepresentation of students in the questionnaire who did not upload their solutions in the live coding. Hence, the results are reported separately for these two groups and aggregated.

7 CONCLUSIONS AND FUTURE WORK

In this paper a scalable teaching format in which students solve small programming assignments during a lecture is described. The e-assessment system GATE was extended to support instructors in getting an overview of all student submissions and common issues faced during the process and to discuss alternative solution strategies. This allows for timely/rapid feedback and targeted intervention that would not be possible otherwise. The e-assessment system provides an overview of the correctness of the submissions, including the most common syntactic and functional errors and their frequencies.

The approach was tested successfully in a first-semester Java course and rated as helpful by the instructor. In a post-term questionnaire, the majority of responding students found the approach valuable and liked it. However, the concrete execution needs improvement, such as allowing more time to work on assignments. It is worth noting that only a small number of students participated in the live coding.

The grouping approach should be further optimized, e. g. filter out "empty" submissions. Apart from the grouping, the system could suggest correct submissions that use different strategies to not only discuss erroneous submissions. Can such an approach also be used to discuss coding style?

This research paves the road for pedagogic questions such as when and how to use it optimally in lectures. Also, the general approach can be used in different scenarios such as in schools, smaller courses, in (group) programming practicals to get a better overview of the progress and common issues, and supporting teaching assistants preparing their exercise sessions.

I thank all the students who participated in the live coding and/or provided feedback in the questionnaire!

REFERENCES

[1] Ella Albrecht and Jens Grabowski. 2020. Sometimes It's Just Sloppiness - Studying Students' Programming Errors and Misconceptions. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20)*. ACM. https://doi.org/10.1145/3328778.3366862

[2] Ryan Baker, Jason Walonoski, Neil Heffernan, Ido Roll, Albert Corbett, and Kenneth Koedinger. 2008. Why students engage in "gaming the system" behavior in interactive learning environments. *Journal of Interactive Learning Research* 19, 2 (2008), 185–224. https://www.learntechlib.org/primary/p/24328/

[3] Jerrold E. Barnett and Alisha L. Francis. 2012. Using higher order thinking questions to foster critical thinking: a classroom study. *Educational Psychology* 32, 2 (mar 2012), 201–211. https://doi.org/10.1080/01443410.2011.638619

[4] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful. In *ITiCSE-WGR '19: Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. ACM. https://doi.org/10.1145/3344429.3372508

[5] James T. Boyle and David J. Nicol. 2003. Using classroom communication systems to support interaction and discussion in large class settings. *Research in Learning Technology* 11, 3 (sep 2003). https://doi.org/10.3402/rlt.v11i3.11284

[6] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101. https://doi.org/10.1191/1478088706qp063oa

[7] Neil C.C. Brown and Amjad Altadmri. 2014. Investigating novice programming mistakes: educator beliefs vs. student data. In *Proceedings of the tenth annual conference on International computing education research (ICER '14)*. ACM. https://doi.org/10.1145/2632320.2632343

[8] Charles H. Chen and Philip J. Guo. 2019. Improv: Teaching Programming at Scale via Live Coding. In *Proceedings of the Sixth (2019) ACM Conference on Learning @ Scale (L@S '19)*. ACM. https://doi.org/10.1145/3330430.3333627

[9] Luca Chiodini, Igor Moreno Santos, Andrea Gallidabino, Anya Tafliovich, André L. Santos, and Matthias Hauswirth. 2021. A Curated Inventory of Programming Language Misconceptions. In *Proc. ITiCSE*. ACM. https://doi.org/10.1145/3430665.3456343

[10] Michael Ebert and Markus Ring. 2016. A presentation framework for programming in programing lectures. In *Proc. EDUCON*. IEEE, 369–374. https://doi.org/10.1109/EDUCON.2016.7474580

[11] Carmen Fies and Jill Marshall. 2006. Classroom Response Systems: A Review of the Literature. *J Sci Educ Technol* 15, 1 (2006), 101–109. https://doi.org/10.1007/s10956-006-0360-1

[12] Daniel Gerhardt, Jan Kammer, Daniel Knapp, Klaus Quibeldey-Cirkel, Christoph Thelen, and Paul-Christian Volkmer. 2013. ARSnova: ein Audience Response System für Inverted-Classroom-Szenarien mit Unterstützung von Just-in-Time Teaching und Peer Instruction. In *Proc. DeLFI 2013*. Gesellschaft für Informatik e. V., Bonn, 297–300.

[13] John Hattie and Helen Timperley. 2007. The Power of Feedback. *Review of Educational Research* 77, 1 (mar 2007), 81–112. https://doi.org/10.3102/003465430298487

[14] Matthias Hauswirth and Andrea Adamoli. 2009. Solve & evaluate with informa: a Java-based classroom response system for teaching Java. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. ACM. https://doi.org/10.1145/1596655.1596657

[15] Niels Heller and Francois Bry. 2019. Organizing Peer Correction in Tertiary STEM Education: An Approach and its Evaluation. *International Journal of Engineering Pedagogy (iJEP)* 9, 4 (2019), 16–32. https://doi.org/10.3991/ijep.v9i4.10201

[16] Niels Heller and François Bry. 2021. Human computation for learning and teaching or collaborative tracking of learners' misconceptions. In *Intelligent Systems and Learning Data Analytics in Online Education*. Elsevier, 323–343. https://doi.org/10.1016/b978-0-12-823410-5.00015-2

[17] Amy Hoyt, John A McNulty, Gregory Gruener, Arcot Chandrasekhar, Baltazar Espiritu, David Ensminger, Ron Price Jr, and Ross Naheedy. 2010. An audience response system may influence student performance on anatomy examination questions. *Anatomical Sciences Education* 3, 6 (2010), 295–299. https://doi.org/10.1002/ase.184

[18] J. Jackson, M. Cobb, and C. Carver. 2005. Identifying Top Java Errors for Novice Programmers. In *Proceedings Frontiers in Education*. IEEE. https://doi.org/10.1109/fie.2005.1611967

[19] Luke Johnston, Madeleine Bonsma-Fisher, Joel Ostblom, Ahmed Hasan, James Santangelo, Lindsay Coome, Lina Tran, Elliott De Andrade, and Sara Mahallati. 2019. A graduate student-led participatory live-coding quantitative methods course in R: Experiences on initiating, developing, and teaching. *Journal of Open Source Education* 2, 16 (June 2019), 49. https://doi.org/10.21105/jose.00049

[20] Robin H Kay and Ann LeSage. 2009. Examining the benefits and challenges of using audience response systems: A review of the literature. *Computers & Education* 53, 3 (2009), 819–827. https://doi.org/10.1016/j.compedu.2009.05.001

[21] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *TOCE* 19, 1, Article 3 (2018). https://doi.org/10.1145/3231711

[22] Alexander Kiy and Sven Strickroth. 2017. Potentiale aufzeigen und Synergien nutzen: Audience Response Systeme als ein Anwendungsgebiet hochschulübergreifender Kooperationen. In *Joint Proceedings of the Pre-Conference Workshops of DeLFI and GMW 2017*, Vol. 2092. CEUR-WS.org, Bonn, Germany. http://ceur-ws.org/Vol-2092/paper11.pdf

[23] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory Programming: A Systematic Literature Review. In *Proc. ITiCSE 2018 Companion*. 55–106. https://doi.org/10.1145/3293881.3295779

[24] Sebastian Mader and François Bry. 2019. Audience Response Systems Reimagined. In *Proc. ICWL*. 203–216. https://doi.org/10.1007/978-3-030-35758-0_19

[25] Sebastian Mader and François Bry. 2019. Fun and Engagement in Lecture Halls through Social Gamification. *International Journal of Engineering Pedagogy (iJEP)* 15, 2 (2019), 117–136. https://doi.org/10.3991/ijep.v9i2.10163

[26] Eric Mazur and Mark D. Somers. 1999. Peer Instruction: A User's Manual. *American Journal of Physics* 67, 4 (apr 1999), 359–360. https://doi.org/10.1119/1.19265

[27] I. T. Chan Mow. 2012. Analyses of student programming errors in Java programming courses. *Journal of Emerging Trends in Computing and Information Sciences* 3, 5 (2012), 739–749.

[28] Lorelli S. Nowell, Jill M. Norris, Deborah E. White, and Nancy J. Moules. 2017. Thematic Analysis: Striving to Meet the Trustworthiness Criteria. *International Journal of Qualitative Methods* 16, 1 (2017). https://doi.org/10.1177/1609406917733847

[29] James Oigara and Jared Keengwe. 2011. Students' perceptions of clickers as an instructional tool to promote active learning. *Education and Information Technologies* 18, 1 (2011), 15–28. https://doi.org/10.1007/s10639-011-9173-9

[30] Archana Pradhan, Dina Sparano, and Cande V. Ananth. 2005. The influence of an audience response system on knowledge retention: An application to resident education. *AJOG* 193, 5 (2005), 1827–1830. https://doi.org/10.1016/j.ajog.2005.07.075

[31] Steven Robbins. 2011. Beyond clickers. In *Proc. SIGCSE*. ACM. https://doi.org/10.1145/1953163.1953347

[32] Marc J. Rubin. 2013. The Effectiveness of Live-Coding to Teach Introductory Programming. In *Proc. SIGCSE*. Association for Computing Machinery, New York, NY, USA, 651–656. https://doi.org/10.1145/2445196.2445388

[33] Stefan Seegerer and Ralf Romeike. 2018. Goals, Topics and Tools of Computer Science for All University or College Courses. In *Proc. SIGCSE*. ACM. https://doi.org/10.1145/3159450.3162237

[34] Ana Selvaraj, Eda Zhang, Leo Porter, and Adalbert Gerald Soosai Raj. 2021. Live Coding: A Review of the Literature. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. ACM. https://doi.org/10.1145/3430665.3456382

[35] Sven Strickroth. [n. d.]. Automatic Grouping of Common Errors in Programming Exercises. (under review). https://www.tel.ifi.lmu.de/software/gate/, the full reference will be posted there, also the source code of GATE is available there.

[36] Sven Strickroth. 2023. Does Peer Code Review Change My Mind on My Submission?. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2023)*. Association for Computing Machinery, 498–504. https://doi.org/10.1145/3587102.3588802

[37] Sven Strickroth. 2023. Towards Live Coding and Instant Feedback on Common Issues in Large Lectures. In *Responsive and Sustainable Educational Futures. 18th European Conference on Technology Enhanced Learning, EC-TEL 2023, Aveiro, Portugal, September 4–8, 2023, Proceedings*. Springer Nature Switzerland, 662–667. https://doi.org/10.1007/978-3-031-42682-7_58

[38] Sven Strickroth and François Bry. 2022. The Future of Higher Education is Social and Personalized! Experience Report and Perspectives. In *Proceedings of the 14th International Conference on Computer Supported Education – Volume 1: CSEDU*, Vol. 1. INSTICC, SciTePress, 389–396. https://doi.org/10.5220/0011087700003182

[39] Sven Strickroth and Florian Holzinger. 2022. Supporting the Semi-Automatic Feedback Provisioning on Programming Assignments. In *Methodologies and Intelligent Systems for Technology Enhanced Learning, 12th International Conference (MIS4TEL'22)*. Springer International Publishing, Cham, 13–19. https://doi.org/10.1007/978-3-031-20617-7_3

[40] Sven Strickroth, Hannes Olivier, and Niels Pinkwart. 2011. Das GATE-System: Qualitätssteigerung durch Selbsttests für Studenten bei der Onlineabgabe von Übungsaufgaben?. In *Proc. DeLFI*. 115–126. https://dl.gi.de/handle/20.500.12116/4740

[41] Sven Strickroth and Michael Striewe. 2022. Building a Corpus of Task-based Grading and Feedback Systems for Learning and Teaching Programming. *International Journal of Engineering Pedagogy (iJEP)* 12, 5 (Nov. 2022), 26–41. https://doi.org/10.3991/ijep.v12i5.31283

[42] Jonas Vetterick, Martin Garbe, and Clemens Cap. 2013. Tweedback: A Live Feedback System for Large Audiences. In *Proc. CSEDU*. https://doi.org/10.5220/0004414501940198

[43] Daniela Zehetmeier, Axel Böttcher, Anne Brüggemann-Klein, and Veronika Thurner. 2015. Development of a Classification Scheme for Errors Observed in the Process of Computer Programming Education. In *HEAd'15. Conference on Higher Education Advances*. https://doi.org/10.4995/head15.2015.356